LLNL-SR-612077

Final Report on Statistical Debugging for Petascale Environments

B. Liblit

January 22, 2013

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Final Report on Statistical Debugging Techniques for Petascale Environments

Professor Ben Liblit
Computer Sciences Department
University of Wisconsin–Madison

October 29, 2012

1 Introduction

Statistical debugging identifies program behaviors that are highly correlated with failures. Traditionally, this approach has been applied to desktop software. In this context, statistical debugging is effective in identifying the causes that underlie several difficult classes of bugs. These include memory corruption, non-deterministic bugs, and bugs with multiple temporally-distant triggers.

The domain of scientific computing offers a new target for this type of debugging. Scientific code runs at massive scales offering massive quantities of statistical feedback data. Data collection can scale well because it requires no communication between compute nodes. Unfortunately, existing statistical debugging techniques impose run-time overhead that is unsuitable for computationally-intensive code. Additionally, the normal communication that occurs between nodes in parallel jobs violates a key assumption of statistical independence in existing statistical models.

In work supported by this subcontract, we have reduce the run-time overhead of statistical debugging instrumentation by up to 20 % over prior work. We have also addressed challenges related to data collection. A paper describing our work in this area has been prepared for submission to a suitable publication venue.

2 Numerically-oriented Instrumentation

Statistical debugging requires information about the run-time behavior of the program being debugged; program behaviors are reduced to *predicates* that evaluate to true or false at certain program points. At any given program point, more than one predicate may exist. An *instrumentation site* is a program point that is instrumented to observe at least one predicate. To reduce the run-time overhead of this data collection, prior work has use a compile-time transformation, the *sampling transformation*, to observe predicates sparsely. For example, only $1/100$ predicates encountered might be observed and recorded. Listing 1 show an example of the sampling transformation.

Notice the conditional check on line 2. If the next predicate to be sampled cannot occur in the current loop iteration, the loop body has no additional run-time checks. While this approach works well for desktop software, it imposes much higher overhead on numerically-intensive code because it (1) adds additional instruction cache pressure, (2) interferes with branch prediction, and (3) can prevent vectorization.

2.1 Optimizations

We have developed an alternative sampling transformation to mitigate these effects and reduce the overhead of sampling in numerically-intensive code. Listing 2 shows an example of our improved approach. While

```
for (i = 0; i < vec_len; i += STRIDE) {
  if (countdown > WEIGHT) {
    // Fast path
  } else {
    // Instrumented path
  }
}
```

Listing 1: An example sampled loop

```
1  i = 0;
2  while (i < vec_len) {
3    int loop_start = i;
4    int bound = vec_len;
5    if (countdown <= (bound - i) / STRIDE * WEIGHT)
6      bound = i + (countdown - 1) / STRIDE * WEIGHT;
7    for (; i < bound; i += STRIDE) {
8      // Completely uninstrumented path
9    }
10   countdown -= (i - loop_start) / STRIDE * WEIGHT;
11   if (i < vec_len) {
12     // Instrumented path
13   }
14   i += STRIDE;
15 }
```

Listing 2: An example optimized loop

the resulting loop is larger, it is much faster than listing 1 because the inner loop on line 7 is completely unmodified. That is, the loop body is identical to the uninstrumented code. This maximizes the amount of dynamically-executed code that takes full advantage of aggressive compiler optimizations including loop unrolling and vectorization. This transformation effectively amortizes the overhead of checking predicates over many loop iterations by only performing checks that will succeed. Our optimization is composable: it applies to loops nested to arbitrary depths.

The loops to which we can apply our optimization are typically inner loops performing numeric operations; few predicates used by our statistical debugging techniques are present in numeric code. Furthermore, these loops execute many times and our techniques do not see a significant benefit from observing a single predicate millions of times instead of just hundreds. These observations led us to a further optimization to fully realize the benefits of the amortized run-time checks: *adaptive sampling*. we exponentially reduce the sampling rate in loops that are executed many times, allowing more iterations to execute before interrupting them to sample a predicate.

2.2 Evaluation

We have evaluated our optimization on several benchmarks. Here we summarize results from two of the Sequoia benchmarks: AMG and IRS. We show the overhead of our optimizations relative to uninstrumented binaries compiled using GCC with standard optimizations. We also show our overhead compared to the prior

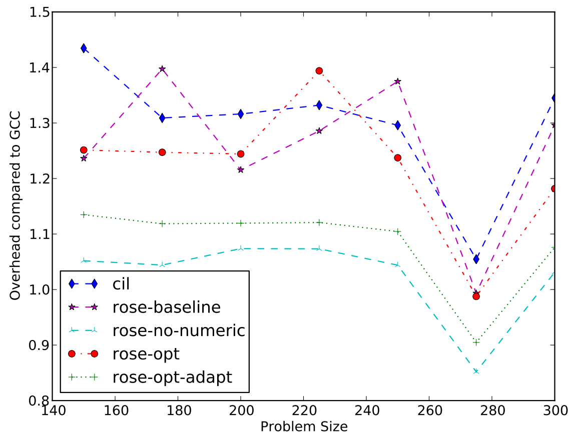

Figure 1: Overhead for AMG

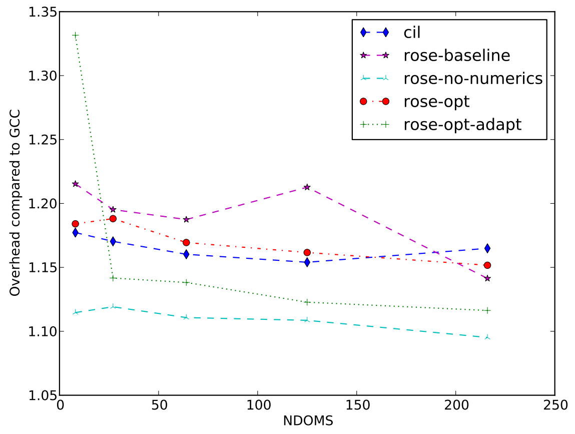

Figure 2: Overhead for IRS

work most similar to ours. This prior work is based on CIL and is labeled cil in the graphs. The rose-baseline entries are binaries produced by our instrumentor with the same sampling transformation as the CIL-based instrumentor. The rose-opt configuration applies our optimized loop transformation over loops nested to a depth of two. The rose-opt-adapt configuration further adds our adaptive sampling transformation on top of rose-opt. For comparison, the rose-no-numeric configuration omits all instrumentation from computational kernels, therefore collecting no debugging information from them. The results are shown in figures 1 and 2. Overall, our optimizations save 20 % overhead for AMG and 5 % for IRS compared to prior work.

3 Scalable Data Collection

Lower run-time overhead is necessary but not sufficient to make statistical debugging viable at large scale. We also need scalable mechanisms to aggregate data from many processes. Prior work relied on the process being debugged to save their own feedback data into the file system as they exited. This poses two problems for scalable debugging. First, a large number of processes reporting reports feedback data at the same time can overload shared file systems, reducing availability for others and possibly losing data. Second, crashing processes are in an unstable state; asking them to report their own data by performing IO can easily result in lost or corrupted feedback reports. Missing or mangled feedback data from crashing processes is highly undesirable for statistical debugging: good feedback from crashing processes is exactly what we need in order to debug faults.

We have addressed both of these problems by creating a scalable feedback data collection framework. Feedback data is transmitted from back-end compute nodes to the front-end node using tree-structured communication provided by the MRNet library. With this service, nearly all data is held in memory and sent over fast network links; only the single front-end node must write data to the file system. We have addressed the second problem by introducing a watchdog process on each compute node to monitor the execution of compute processes via the ptrace debugging interface (through StackWalkerAPI). Each compute process writes its feedback data into a shared memory segment that is also mapped by its watchdog process. When the compute process terminates (either normally or due to a crash), feedback data in the shared memory segment persists and the watchdog reports it by sending it into the communication tree. This is much more robust than previous methods because feedback data can be reported no matter how the process being debugged crashes. Data can be retrieved even if the process was terminated with a SIGKILL.

Individual feedback reports can be large; in massively-parallel applications, the amount of data reported can be enormous. We have experimented with several approaches and data encodings to make the feedback

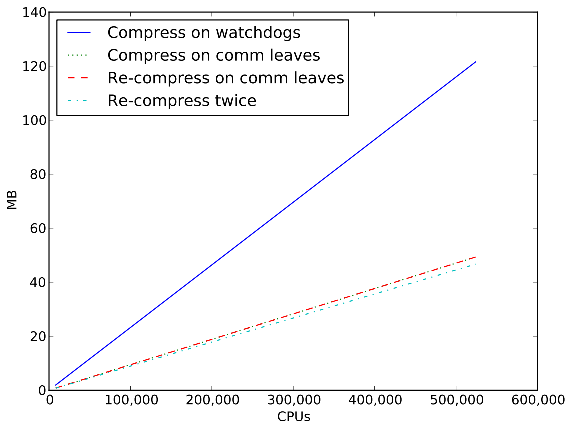

Figure 3: Feedback data sizes

data size manageable. We discuss LZMA compression-based methods here. Figure 3 shows feedback sizes at a variety of process counts and using a variety of compression strategies. In the first configuration, we compressed feedback data only on watchdog nodes. In the second, we only compress the data at the leaves of the communication tree, each of which receives data from 32 watchdog processes. In the other configurations, data is uncompressed and recompressed at higher levels in the communication tree. Earlier compression results in lower CPU loads higher in the communication tree, but more data sent over the network at the leaves. On the other hand, later compression can exploit more redundancy between reports. Our simulations with randomly perturbed feedback reports indicate that compressing data at the leaves of the communication tree would allow us to collect the feedback data from 500,000 processes with less than 50 MB of data transferred to the front-end node.

4 Implementation Artifacts

Our source-to-source instrumenting compiler, which inserts instrumentation points into programs being debugged, builds on the ROSE compiler infrastructure, developed at LLNL. This instrumentor extends support for sampled instrumentation to C++ and Fortran. (C was already supported by prior work.) In the process of writing this new instrumentor, we submitted many bug reports and patches to the ROSE project.